Springboot总结

使用MyBatis访问数据库

1. 在`pom.xml`中添加依赖库，第一项为springboot-web必备项，第二项为mybatis提供的与springboot连接的组件，第三项为mysql支持，第四项用于测试

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>2.1.2</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

2. 在`src/main/resources/application.properties`中配置数据源：

```
server.port=8333
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/learnspringboot?
serverTimezone=UTC&characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Diver
```

3. 定义对应的POJO

```
public class User {
    private int id;
    private String name;
    private int age;
    private double money;
    //Setter and Getter
}
```

4. 定义DAO，使用了注解@Mapper

```
@Mapper //这了使用了注解@Mapper
public interface UserDao {
    //通过名字查询用户信息
    @Select("SELECT * FROM user WHERE name = #{name}")
    User findUserByName(@Param("name") String name);

    //查询所有用户信息
    @Select("SELECT * FROM user")
    List<User> findAllUser();

    //插入用户信息
    @Insert("INSERT INTO user(name, age,money) VALUES(#{name}, #{age}, #
{money})")
    void insertUser(@Param("name") String name, @Param("age") Integer age,
@Param("money") Double money);

    //根据 id 更新用户信息
    @Update("UPDATE  user SET name = #{name},age = #{age},money= #{money}
WHERE id = #{id}")
    void updateUser(@Param("name") String name, @Param("age") Integer age,
@Param("money") Double money,
                    @Param("id") int id);

    //根据 id 删除用户信息
    @Delete("DELETE from user WHERE id = #{id}")
    void deleteUser(@Param("id") int id);
}
```

5. 定义Service，使用注解@Service

```
@Service
public class UserService {
    @Autowired
    private UserDao userDao; //这里有一个奇怪的bug，出现的原因是
@SpringBootApplication配置和Intellij的版本未能完全兼容，在实际运行中并不会有问题

    //根据名字查找用户
    public User selectUserByName(String name) {
        return userDao.findUserByName(name);
    }

    //查找所有用户
    public List<User> selectAllUser() {
        return userDao.findAllUser();
    }


    //插入两个用户
    public void insertService() {
        userDao.insertUser("SnailClimb", 22, 3000.0);
        userDao.insertUser("Daisy", 19, 3000.0);
    }

    //根据id 删除用户
    public void deleteService(int id) {
        userDao.deleteUser(id);
    }
```

```
    //模拟事务。由于加上了 @Transactional注解，如果转账中途出了意外 SnailClimb 和
Daisy 的钱都不会改变。
    @Transactional
    public void changemoney() {
        userDao.updateUser("SnailClimb", 22, 2000.0, 3);
        // 模拟转账过程中可能遇到的意外状况
        int temp = 1 / 0;
        userDao.updateUser("Daisy", 19, 4000.0, 4);
    }
}
```

6. 定义Controller

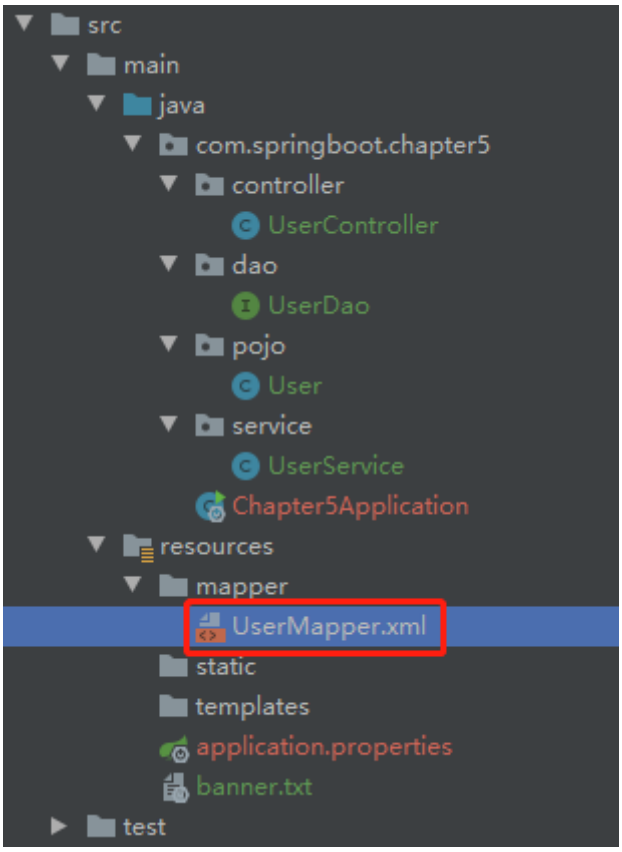
```
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired //自动装配Service
    private UserService userService;

    @RequestMapping("/query")
    public User testQuery() {
        System.out.println("in!");
        User daisy = userService.selectUserByName("Daisy");
        System.out.println(daisy);
        return daisy;
    }

    @RequestMapping("/insert")
    public List<User> testInsert() {
        userService.insertService();
        return userService.selectAllUser();
    }

    @RequestMapping("/changemoney")
    public List<User> testchangemoney() {
        userService.changemoney();
        return userService.selectAllUser();
    }

    @RequestMapping("/delete")
    public String testDelete() {
        userService.deleteService(3);
        return "OK";
    }

    @RequestMapping("/testAllQuery")
    public List<User> testAllQuery() {
        return userService.selectAllUser();
    }
}
```

7. Springboot启动项

```
@MapperScan("com.springboot.chapter5.dao")//这里可选，可加上Mapper扫描路径说明
public class Chapter5Application {
    public static void main(String[] args) {
        SpringApplication.run(Chapter5Application.class, args);
    }
}
```

8. 方法②之使用xml文件进行sql编写：

在`resources`路径下新建一个`mapper`文件夹，增加一个`*mapper.xml`文件：



改写`UserDao.java`文件中的函数，比如下面这个根据人名查找信息的函数：

```
@Mapper
public interface UserDao {
    //通过名字查询用户信息，与方法一中的版本不一样在于这里没有了由注释写的SQL语句
    User findUserByName(String name);
}
```

在上面的`UserMapper.xml`中增添内容，注意要明确好`namespace`和`id`以及`resultType`：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.springboot.chapter5.dao.UserDao">
    <select id="findUserByName" parameterType="String"
resultType="com.springboot.chapter5.pojo.User">
        SELECT * FROM user WHERE name = #{name}
    </select>
</mapper>
```

最后一定要记得在`applicationContext.properties`中增加下面的设置：

```
mybatis.mapper-locations=classpath:mapper/*.xml
```

9. 别名Alias问题：

   可以在POJO类上的添加`@Alias(value = "xxx")`指定别名，然后在`applicationContext.properties`上添加

   ```
   mybatis.type-aliases-package=com.springboot.chapter5.pojo
   ```

   记得只需要指定到POJO这个包一层即可。然后在`UserMapper.xml`里面的SQL中，就可以直接使用对应的别名，来减少复杂程度：

   ```
   <select id="findUserByName" parameterType="String" resultType="User">
           SELECT * FROM user WHERE name = #{name}
       </select>
   </mapper>
   ```

10. a
11. a
12. a
13. a
14.

使用JPA访问数据库

1. **添加依赖库**

   在`pom.xml`中

   ```
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>
   <dependency>
       <groupId>mysql</groupId>
       <artifactId>mysql-connector-java</artifactId>
   </dependency>
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-jdbc</artifactId>
   </dependency>
   ```

2. **配置数据源**

   在`src/main/resources/application.properties`中。注意在某些版本中需要添加时区设置，如下

```
spring.datasource.password=admin
spring.datasource.username=root
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/learnspringboot?
serverTimezone=UTC&characterEncoding=UTF-8
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.hibernate.ddl-auto=none
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
spring.jpa.show-sql=true
```

3. **定义POJO**

```
@Entity                                                    //标明是实体类POJO
@Table(name = "springboot2x")                               //与数据库中对应的表
名一致
public class UserForTest {
    @Id                                                    //主键
    @GeneratedValue(strategy = GenerationType.IDENTITY)      //主键递增策略
    private Long id = null;

    @Column(name = "name")                                  //定义属性和表的映射
关系
    private String name;
    @Column(name = "note")
    private String note;
    @Convert(converter = SexEnumConverter.class)             //定义转换器，用于
自定义类和表的映射
    private SexEnum sex;
    /*getter & setter & toString...*/
}
//-------------------------------------------------------------------------
----------------------
public enum SexEnum {                                      //性别枚举类
    MALE(1,"man"),
    FEMAILE(2,"woman");

    private int id;
    private String name;

    SexEnum(int id, String name){
        this.id = id;
        this.name = name;
    }
    public static SexEnum getEnumById(int id){
        for(SexEnum sex : SexEnum.values()){
            if (sex.id == id){
                return sex;
            }
        }
        return null;
    }
```

4. **上文定义的Converter**

```
public class SexEnumConverter implements
AttributeConverter&amp;lt;SexEnum,Integer&amp;gt; {
    @Override
    public Integer convertToDatabaseColumn(SexEnum attribute) {
        return attribute.getId();
    }

    @Override
    public SexEnum convertToEntityAttribute(Integer dbData) {
        return SexEnum.getEnumById(dbData);
    }
}
```

4. **定义JPA接口（DAO / Mapper）**

```
public interface JpaUserForTestRepository extends JpaRepository<UserForTest,
Long> {
}
```

5. **定义Service层**

6. **定义Controller**

```
@Controller
@RequestMapping("/jpa")
public class JpaController {
    @Autowired
    private JpaUserForTestRepository userForTestRepository = null;  //一般开
发会是Service，而不是Dao

    @RequestMapping("/getUserForTest")
    @ResponseBody                                                   //直接返回
json数据用于测试
    public UserForTest getUserForTest(Long id){
        UserForTest userForTest = userForTestRepository.findById(id).get();
        return userForTest;
    }
}
```

7. **Springboot启动文件**

```
@SpringBootApplication
//启用JPA编程，定义JPA接口扫描包路径
@EnableJpaRepositories(basePackages = "springboot.chapter2.dao")
//定义实体Bean/Entity/POJO扫描包路径
@EntityScan(basePackages = "springboot.chapter2.pojo")
public class Chapter2Application {
    public static void main(String[] args) {
        SpringApplication.run(Chapter2Application.class, args);
    }
}
```

8. **访问浏览器对应路径并带上参数**

```
http://localhost:8090/jpa/getUserForTest?id=1
```

9. **自定义查询语句**

   只需要按照一定规则来命名方法，就可以在不写任何代码的情况下完成逻辑。直接指定返回类型则可以规定查询结果的类型（是单条数据还是一组数据），具体如下

```
/*jpa增加实现方法*/
public interface JpaUserForTestRepository extends JpaRepository<UserForTest,
Long> {
    List<UserForTest> findByNameLike(String name);
    List<UserForTest> getUserForTestsByIdAfter(Long id);
    void deleteUserForTestByIdEndsWith(Long id);
}
//------------------------------------------------------------------------------
----------------------

/*具体调用的controller*/
@RequestMapping("testMoreJpa")
@ResponseBody
public List<UserForTest> testMoreJpa(/*Long id*/ String like ){
    //List<UserForTest> userForTestsByIdAfter =
userForTestRepository.getUserForTestsByIdAfter(id);

    /*这里要注意like的前后需要拼接上百分号%*/
    List<UserForTest> byNameLike =
userForTestRepository.findByNameLike("%"+like+"%");
    System.out.println(byNameLike);
    return byNameLike;
}
```

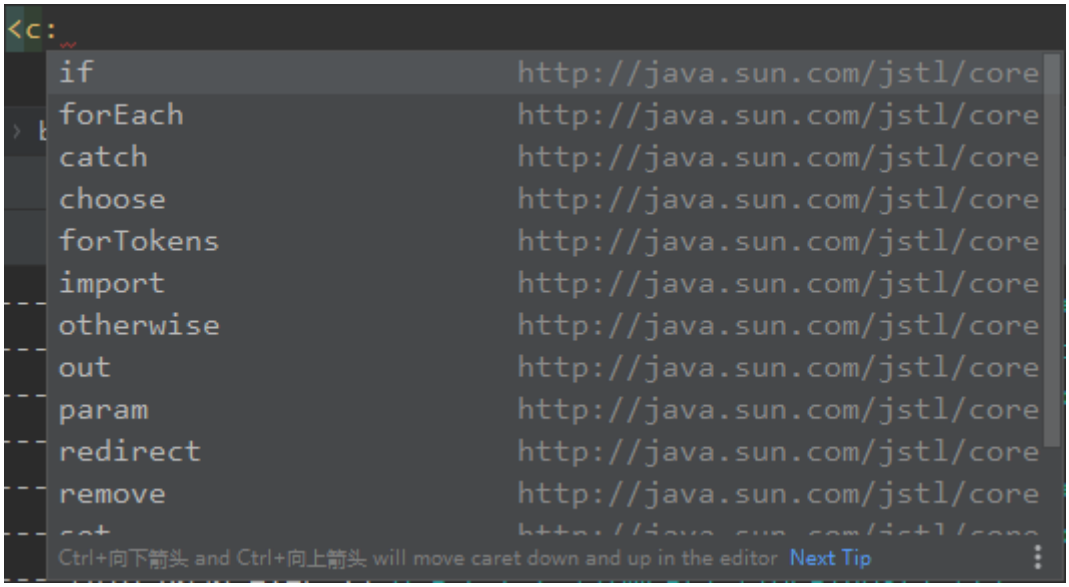
JSP数据接收及显示

在controller这边可以使用Model参数进行数据的传输：

```
@RequestMapping("getUsers")
public String getUsers(Model model){
    model.addAttribute("name","n1");
    model.addAttribute("age","18");

    List<User> users = new ArrayList<>();
    users.add(new User((long) 1,"name1","note1"));
    users.add(new User((long) 2,"name2","note2"));
    users.add(new User((long) 3,"name3","note3"));
    model.addAttribute("users",users);

    return "index"; //这里会索引到index.jsp页面中，并在该页面中进行值的获取
}
```

紧接上面来说，需要在index.jsp文件开头添加这个jstl支持，以使用例如 `c:forEach` 等功能：

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```



比如我们要将上面插入的users列表进行获取并展示，则可以在index.jsp中弄一个表格：

```
<div>
    <table>
        <thead>
        <tr>
            <td>ID</td>
            <td>UserName</td>
            <td>Note</td>
            <td>Item</td>
        </tr>
        </thead>
        <tbody>
        <c:forEach items="${users}" var="item">
            <tr>
                <td>${item.id}</td>
                <td>${item.userName}</td>
                <td>${item.getNote()}</td>
                <td>${item}</td>
            </tr>
        </c:forEach>
        </tbody>
    </table>
</div>
```

对于那些前后端分离的应用来说，我们可能会使用到Vue，如要使用Vue，则要添加这行：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.18/vue.min.js">
</script>
```